OOMTool : AN OMT BASED TOOL FOR OBJECT ORIENTED SOFTWARE DEVELOPMENT

by

K. SIVA PRASADA REDDY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1996

OOMTool: AN OMT BASED TOOL FOR OBJECT ORIENTED SOFTWARE DEVELOPMENT

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

K.Siva Prasada Reddy

to the

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

January 1996.

CERTIFICATE

This is to certify that the work contained in the thesis entitled *"OOMTool:An OMT Based Tool for Object Oriented Software Development"* by "*K.Siva Prasada Reddy*", has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Prof. RMK.Sinha
Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

Dr. Harish Karnick
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

Acknowledgements

I am grateful to my thesis supervisors, Dr. Harish Karnick and Dr. R.M.K. Sinha for their constant support, guidance and encouragement throughout this work. Dr. R.M.K.Sinha introduced me to the interesting field of Object Oriented systems and suggested me this work. Whenever I faced problems Dr. Harish Karnick helped me out. The whole credit of ensuring the timely completion of this thesis goes to Dr. Harish Karnick.

I thank to the class of M.Tech. '94 for making my stay at IITK, a memorable period. Special thanks to my wing mates(HALL IV) MLPK.Reddy, MR.Lakshmi Narayana, Sricharan, Srivatsa for their help and for the *SOLLU* (slang for chitchating) we had together. I also thank Killi for proofreading this thesis.

Finally, I thank my parents and brother for their constant encouragement.

K. Siva Prasada Reddy
1/1/1996.

Abstract

In this report we describe the design, development and implementation of OOMTool which helps the software developers in developing the software using Rumbaugh's Object Modeling Technique (OMT). This tool consists of two parts: 1. Editors for the Object and Dynamic models of OMT, 2. C++ code generator. The Editors provide the users with the facility to draw, save and read the Object and Dynamic models. The user interface for these Editors is window based. The C++ code generation module generates the C++ class headers, required to implement the design. Some elements of the Object model have a direct mapping to C++ (e.g. OMT *object* maps to C++ *class*, Generalization maps to Inheritance etc.). But OMT's fundamental elements association and aggregation have no direct mapping in C++. Consequently, a user has to implement these elements. This code generation module maps these elements using the C++ template based mechanism which satisfies all constraints specified in the Object model. The generated code can be elaborated by the user to get a complete implementation. The editors generate and use a textual intermediate representation which can be used to generate code for other platforms (e.g. RDBMS or other languages).

Contents

Chapter 1

Introduction

1.1 Introduction

A Software Enginering Methodology is a process for the organized production of Software using a collection of predefined and notational conventions. Currently, Structured methodology and Object Oriented methodology are two well defined methodologies which are used by software developers. These two methodologies mainly differ in giving the priority to different views of the system. Normally, a Software system will be seen in three orthogonal views. These views are: static structural description, time dependent dynamic description and description of data transformation from input to output.

The structured methodology gives importance to the dynamic nature of the system where as the Object Oriented Methodology(OOM) gives importance to the structure of the system. OOM is gaining importance because the structure of a system normally changes less frequently than dynamic nature of the system. e.g. The objects in a banking system today will probably be the same as the objects in a banking system five years from now, but the functions and procedures for manipulating those objects may have changed radically.

1.2 Motivation

Over the past few years numerous methodologies have been introduced for Object Oriented Software development. Rumbaugh's Object Modeling Technique(OMT) is currently one of the most popular analysis and design methodologies. This OMT provides developers a diverse and powerful way to design Object Oriented Systems. Some of the other OO methodologies are Booch[3], Coad Yourdon[2] etc.

As mentioned in the above section any software development methodology, in general will view the developing system in three orthogonal views. In OMT these three orthogonal views correspond to Object model, Dynamic model and Functional model. The OMT Object Model is a fast, intuitive approach for identifying and modeling all the objects comprising a system. Details, such as class attributes, methods, inheritance, and associations can also be easily expressed. The dynamic behavior of objects within a system can be described using the OMT Dynamic Model. This model lets you specify detailed state transitions and their descriptions within a system. Finally, process descriptions, and producer/consumer relationships, can be expressed using OMT's Functional Model. Overall, the Rumbaugh/OMT methodology provides one of the strongest tool sets for the analysis and design of object-oriented systems.

In addition to the Methodology used, usage of the Tools will also have a significant impact on the software development process and the quality of software developed.

For analysis and design methodologies that are based on formally defined modeling concepts, it is possible to automatically translate analysis/design models into implementations. This would eliminate the expensive and error-prone step of *manual* translation into a programming language implementation and would, thereby, improve the reliability of the implementation and the productivity of the development team.

This thesis is an attempt to build a Tool which can help in all models of Rumbaugh Object Modeling Technique and produce the C++ code as much as possible (produce C++ class headers to all objects while also producing some methods in each class, to maintain all the relationships between the objects).

1.3 Features of OOMTool

The OOMTool provides full, integrated support for the first two OMT models (Object Model and Dynamic Model), while also providing detailed generation of C++ code for the class structure information found in the OMT Object Model. The OOMTool drawing capabilities and the online help let the user create his OMT designs faster and accurately. As a result user will be able to remain focused on designing systems, not the tedious mechanics of how to draw them.

1.3.1 Object Model

OOMTool supports most of the constructs used with the OMT Object Model. Constructs such as object attributes and methods can be easily specified. Full method prototyping is supported. Associations are also supported and can be implemented in one of several ways, depending upon the multiplicity, link attributes role names and unifiers. Single and multiple inheritance are fully supported, generating appropriate class and subclass structures in generated C++ source.

1.3.2 Dynamic Model

Support for the OMT Dynamic Model is also detailed and comprehensive. Complete specification of event attributes, output events, and action/activity constructs can be represented. State generalization (nesting of states) for multy level description of the state and dynamic diagram iconification for supporting the concurrent dynamic diagrams and external events between them, are provided. From the above descriptions one can generate very good comments(documentation) in the source code generated by OOMTool.

1.3.3 C++ code generation

The OOMTool generates detailed C++ code from OMT Object Model diagrams. When generated, this code contains all the necessary information for implementing the structural aspects of classes, as well as the details of their inheritance and associations. To maintain the different requirements a

library is provided with this tool. This library contains various C++ class definitions required to maintain different aspects (e.g: multiplicity, qualifiers etc.) of the system.

1.4 State of the Art

In last few years, many tools of this kind have been developed for different methodologies. Some of these tools are commercially available e.g.

- *Graphical designer*: provides full, integrated support for all three OMT models. It is developed by Advanced Software Technologies Ltd. One can get the info from http://www.csn.net/infopage
- *ObjecTime*: is an object-oriented CASE tool developed specifically for the analysis, design, verification and implementation of complex distributed real-time systems. It originated in Bell-Northern Research (BNR). ObjecTime supports an advanced methodology for the analysis and design of distributed, event-driven systems known as Real-Time Object-Oriented Modeling (ROOM). ROOM employs graphical design concepts and a highly-iterative development process to help eliminate error-prone discontinuities between the various phases of software development. Information can be obtained from Garth Gullekson, ObjecTime Limited, 1-800-567-TIME, sales@objectime.on.ca
- *BOCS*: is used to create programming language independent specifications, then automatically generate formatted documentation combining text and graphics into popular publishing packages. BOCS also provides code generation for C++ and smalltalk
  info is available at http://www.qucis. queensu.ca/Software Engineering/blurb/bridgepoont.html This tool is from Objective Spectrum.

1.5 Organization of the thesis

The remainder of this thesis is organized into 4 chapters and 5 appendixes.

- Chapter-2 gives an overview of Rumbaugh's Object Modeling Technique and it describes OMT using an example.

- Chapter-3 describes the user interface part of the implemented tool.
- Chapter-4 presents the various steps in developing this OOMTool i.e it discusses the analysis, design and implementation of this OOMTool.
- Chapter-5 gives the conclusions and some hints for further work.
- The Appendix A contains the User manual. Rest of the four appendixes contains brief C++ headers for implementing this OOMTool, C++ code generated by the OOMTool for the example explained in chapter-2, C++ headers for the template library that is provided along with this tool, and format for intermediate code of Object Diagram.

Chapter 2

Object Modeling Technique

2.1 Methodology

Rumbaugh's Object Modeling Technique methodology consists of building a model of an application domain and then adding implementation details to it during the design of the system. It will also describe the graphical OMT notation for representing OO concepts. There are four stages in the system development. These stages are 1. Analysis 2. System design 3. Object design 4. Implementation

Object Oriented concepts can be applied through out the system development life cycle from analysis through design to implementation. The same classes can be carried from stage to stage without a change of notation, although they gain additional implementation details in later stages.

This OMT uses three kinds of models to describe a system: Object model, Dynamic model and Functional model. Rest of this section describes various concepts and OMT notion for these three models.

2.1.1 Stages in OMT

Analysis

Starting from a statement of the problem, the analyst builds a model of the real world situation showing its important properties. The analysis model is

a concise, precise abstraction of what the desired system must do, not how it will be done.

System design

The system designer makes high level decisions about the overall architecture. During the system design the target system is organized into subsystems based on both the analysis structure and the proposed architecture.

Object design

The Object designer builds a design model based on the analysis model but containing implementation details. The designer adds details to the design model in accordance with the strategy established during system design. The focus of the object design is the data structures and algorithms needed to implement each class.

Implementation

The object classes and relationships developed during object design are fully translated into a particular programming language, database or hardware implementation. *Programming should be a relatively minor and mechanical part of the development cycle, because all of the hard decisions should be made during design.*

2.1.2 Object Model

It captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize each class of objects. This structure is shown using the diagrams. This Object diagram is a graph whose nodes are object classes and whose arcs are relationships among classes.

Components of the Object diagram are:

Objects and Classes

OBJECT: is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes:They promote understanding of the real world and provide a practical basis for computer implementation.
CLASS: describes a group of objects with similar properties(attributes), common behavior(operations), common relationships to other objects and common semantics. Most objects derive their individuality from differences in their attribute values.

The OMT symbol for Class is

Class-Name
dataType1: attribute1,attribute2; dataType2: attribute3,attribute4; ....

resultType1 operationName1(argList);
resultType2 operationName2(argList);
....

Figure 2.1 Notation for CLASS

An attribute is a data value held by the objects in a class. An operation is a function that may be applied to or by the objects in a class

Links and Associations

LINK: is a physical or conceptual connection between object instances.
ASSOCIATION: describes a group of links with common structure and common semantics. Associations are inherently bidirectional. Associations can have many attributes. Some of these are

- MULTIPLICITY: specifies how many objects of one class may relate to a single instance of an associated class. There are different types of multiplicity that can be mentioned for one end of an association: optional, one, many, range(n+), set(2,4,6) etc. The graphical representation used by OMT to depict multiplicities is given below.

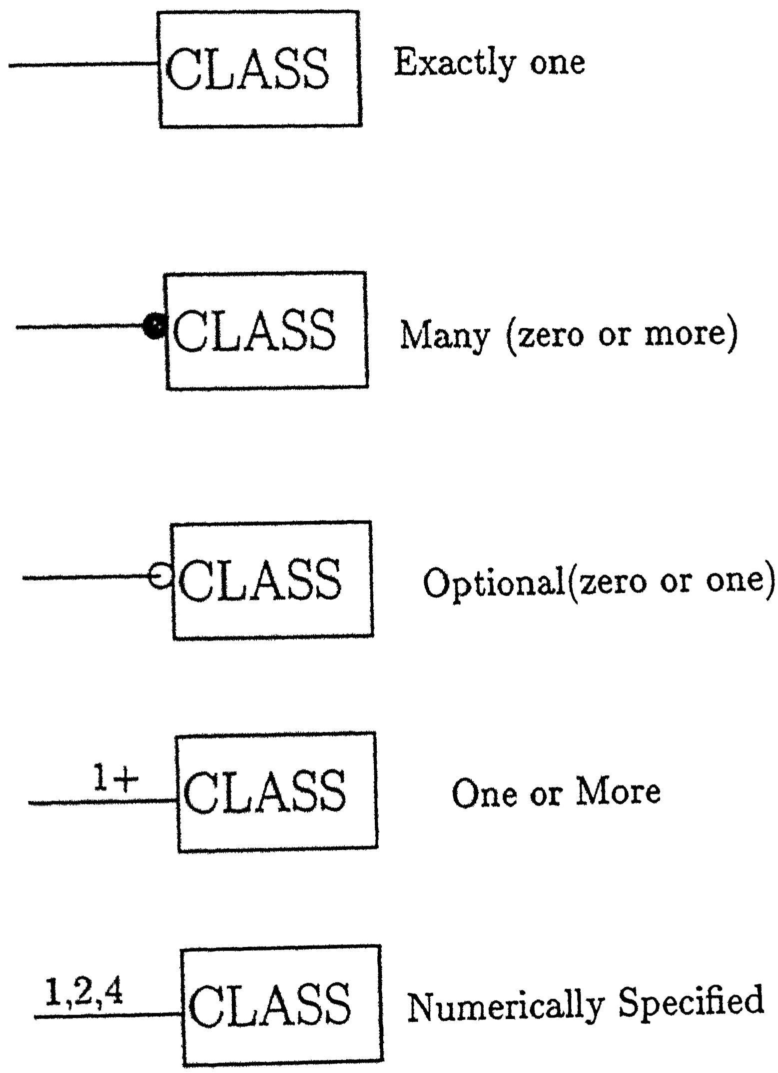


Figure 2.2 Multiplicity of Associations

- QUALIFIER: is a special attribute that reduces the effective multiplicity of an association. The qualifier distinguishes among the set of objects at the *many* end of an association.

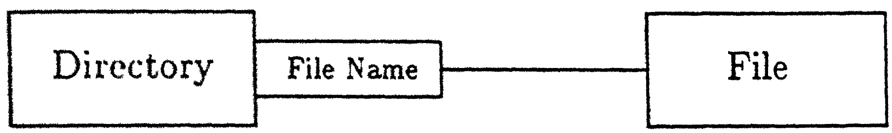


Figure 2.3 A qualified association

- LINK ATTRIBUTE: is a property of the link in an association. Each link attribute has a value for each link.

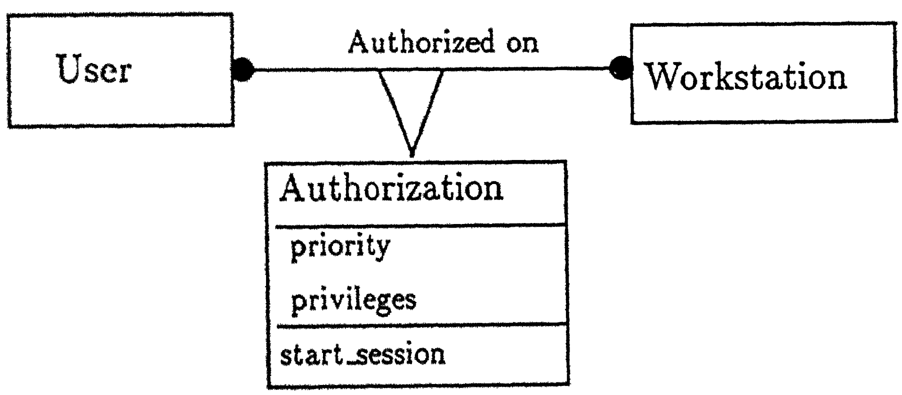


Figure 2.4 Link attribute and Modelling an association as a class

The above figure shows the authorization information for users on workstations. This authorization information is the attribute of neither User nor Workstation. It is modeled as a separate class.

- ROLE NAMES: A role is one end of an association. A binary association has two roles, each of which may have a role name. A role name is a name that uniquely identifies one end of an association.

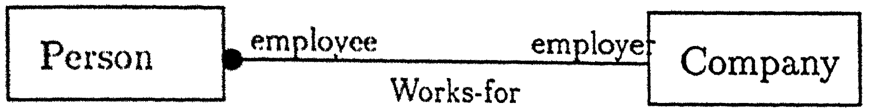


Figure 2.5 Role names for an association

Role names are necessary for associations between two objects of the same class.

Aggregation

Aggregation is the "part_whole" or "a_part_of" relationship in which objects representing the components of something are associated with an object representing the entire assembly. Aggregation is antisymmetric, that is if B is part of A, then A is not part of B. We define an aggregation relationship as relating an assembly class to one component class. An assembly with many kinds of components corresponds to many aggregation relationships. We define each individual pairing as an aggregation so that we can specify the multiplicity of each component within the assembly. This definition emphasizes that aggregation is a special form of association. Aggregation is drawn like association, except a small diamond indicates the assembly end of the relationship.

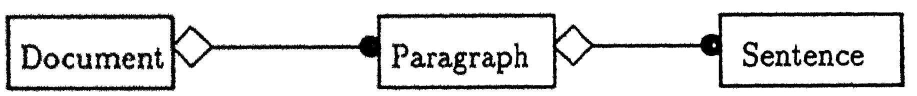


Figure 2.6 Aggregation

A document consists of many paragraphs, each of which consists of many sentences.

Generalization and Inheritance

Generalization and inheritance are powerful abstractions for sharing similarities among classes while preserving their differences. Generalization is the

relationship between a class and one or more refined versions of it. Each subclass is said to inherit the features of its super class.

Generalization and inheritance are transitive across multiple levels. An instance of subclass is simultaneously an instance of all its ancestor classes. The state of an instance includes a value for every attribute of every ancestor class. Any operation on any ancestor class can be applied to an instance.

The OMT notation for generalization is a triangle symbol connecting a superclass to its subclasses.

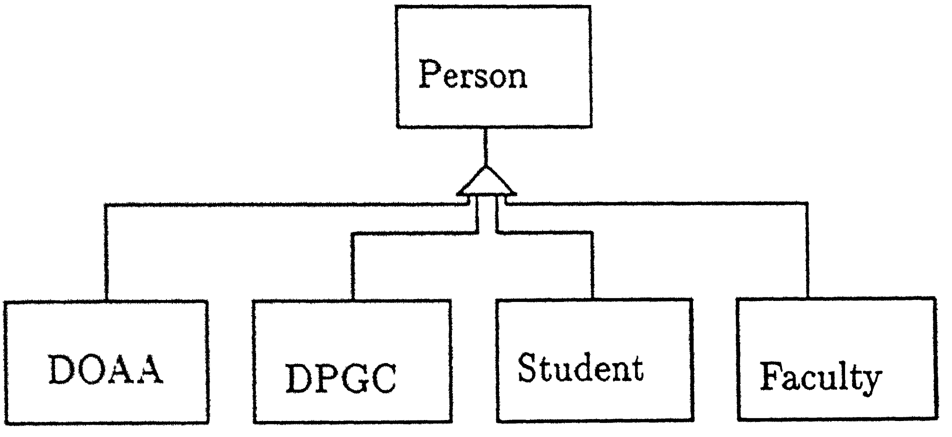


Figure 2.7 Generalization

Multiple inheritance: A class may inherit from more than one class.

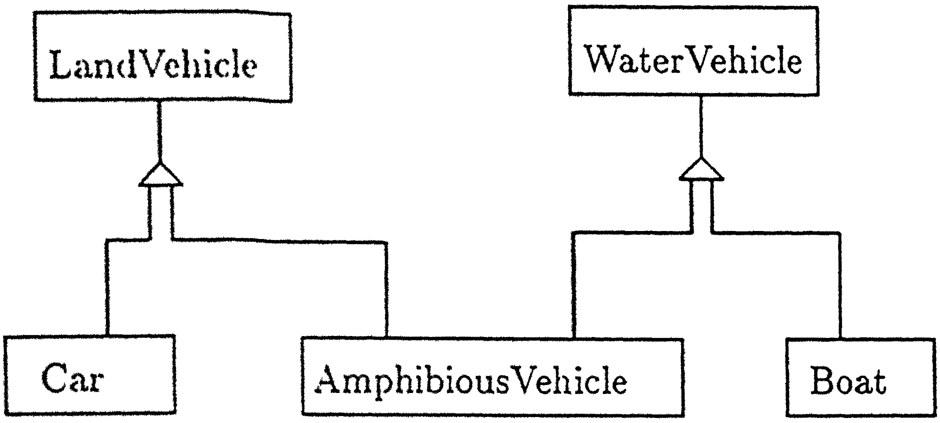


Figure 2.8 Multiple inheritance

There are more concepts which are not being described here since they are rarely used (e.g. derived objects, constraints on objects etc.). For detailed information on these concepts, reader may refer to [1].

2.1.3 Dynamic Model

The dynamic model describes those aspects of a system that are concerned with time and change. Control is that aspect of a system that describes the sequences of operations that occur in response to external stimuli, without consideration of what the operations do, what they operate on, or how they are implemented. The major dynamic modeling concepts are events, which represent external stimuli, and states, which represent values of objects. Using these two concepts(states, events), dynamic model diagram can be constructed. It is a graph whose nodes are states and whose arcs are events. In literature this is usually referred to as a state diagram.

State

The attribute values and links of an object are called its STATE. Over time Objects change their state.

Event

An individual stimulus from one object to another is an event. The response to an event depends on the state of the object receiving it, and can include a change of state or the sending of another event to some other object.

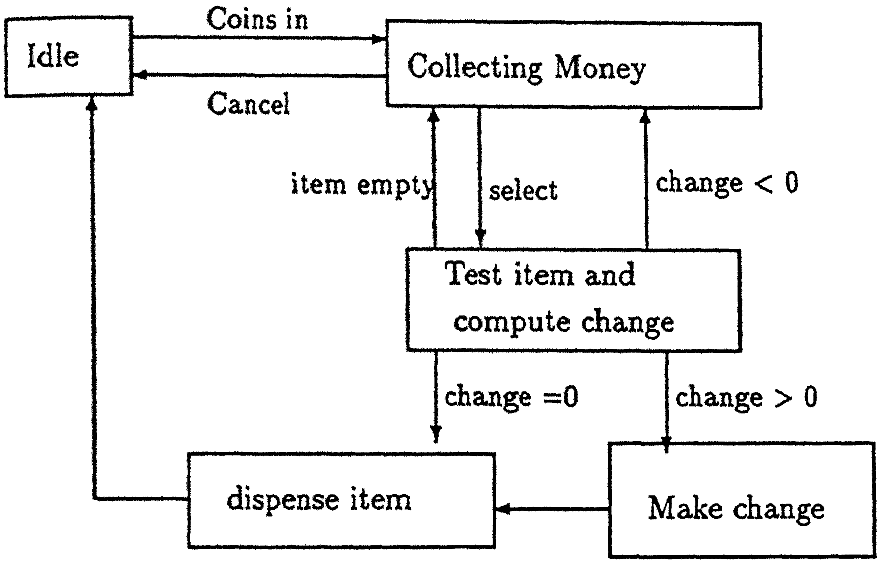


Figure 2.9 Dynamic diagram for Vending Machine

For detailed information on these concepts, reader may refer to [1].

2.1.4 Functional Model

The functional model shows how output values in a computation are derived from input values, without regard to the order in which the values are computed. The functional model consists of multiple data flow diagrams which show the flow of values from external inputs, through operations and internal data stores, to external outputs. The functional aspect has been deemphasized in this thesis and we will not discuss it any further.

For more detailed information on these concepts, reader may refer to [1].

2.2 Example: Dean Of Academic Affairs(DOAA)

In this section we illustrate OMT using the automation of DOAA's office automation as an example.

2.2.1 Problem Statement

Structure: The whole institute is a set of departments. Each department consists of Faculty members, Students. The DOAA office is a centralized system to administer and co-ordinate the academic functionality of the institute. At the departmental level academic administration is handled by two committees: DUGC(for UG's) and DPGC(for PG's).

This system has to mantain all academic information:

- Student details
  - Current Students information
  - Old students information(History)
- Department information
  - registered students
  - courses offered
- History of each Course
- Faculty information
  for each faculty member
  - bio-data
  - semesterwise commitments
- Rules for different aspects
  - Grading policy
  - minimum CPI
  - minimum duration etc.

It has to do the following functions:

- Registration process: provide each student
  - registration sheet
  - backlogs
  - eligible courses list
- Grade processing and printing Grade sheets by applying the corresponding rule depending on the student.
- Respond to various queries
  - attendance lists
  - generation of various lists of students depending on the type of student (e.g. SC, ST, BC, Exchange, Foreign etc.).
  - printing specific student details such as bio-data, backlogs, leave details, punishment, thesis details etc.
  - printing semesterwise department details
  - printing semisterwise faculty commitments.
- Provide interfaces to all the users to give above queries
- Protect the information from modifying or viewing by unauthorized users
  - No student can query other's details etc.
- Provide Communication facility between the users.

2.2.2 Objects Identified

1. Users
   - DOAA
   - DPGC
   - Student

- Faculty

2. History
    - Students History
    - Department History
    - Faculty History
    - Courses History
3. Rules
4. Mailing system

2.2.3 Attributes and Methods of each object

Here we will identify vital attributes and methods of each object.

DOAA

attributes: name, passwd, etc.
methods: interface through which DOAA can do the following functic

- feeding information
    - student details
    - department details
    - faculty details
    - details of courses
    - rules for various calculations etc.
- various enquiries and commands
    - generation of student lists depending on the type of studen ;. SC, ST, BC, Exchange, Foreign etc.).
    - enquiries about a specific student. (e.g. bio-data, backlogs e details, punishment, thesis details etc.).

- printing attendance lists.
- enquiries about the department, faculty and courses.
- print grade cards
- start registration process

- mailing system: using this DOAA can interact with the other users.

DPGC

attributes: name, passwd, deptName, etc.
methods: interface to DPGC to perform the following

- enquiries: restricted to the department
  - generation of student lists depending on the type of student. (e.g. SC, ST, BC, Exchange, Foreign etc.).
  - enquiries about a specific student. (e.g. bio-data, backlogs, leave details, punishment, thesis details etc.).
  - enquiries about the faculty and courses.
- mailing system

Faculty

attributes: name, passwd, deptName, etc.
methods: interface

- show course details
- mailing system etc.

Student

attributes: name, passwd, deptName, etc.
methods: interface

- request and submit registration form

Rules

attributes: version, applicableTo,rule etc.
methods:

- calculate CPI for a given student
- print grade sheet
- check minimum CPI
- check minimum duration
- validate a student for awarding the degree

2.2.4 Associations, Generalizations, Aggregations between objects

1. Generalizations: We can have an object called "Person" which is a generalization of DOAA, DPGC, Faculty, Student.
2. Aggregations: We can have one more object called "History" which is an aggregation of Student, Dept, Faculty, Courses histories.
3. Associations: Object Person will have the association(using relation) with mailing system.
   Object Person will have the association with History object.
4. *Rules* object is an aggregation of *Rule* objects. As rules are needed to evaluate different things in all objects, we keep this *Rules* object as a global object.

These above things are in high level and some more objects may come at the implementation level

2.2.5 Object Diagram

The following diagram illustrates the high level design. For clarity purposes we are not giving the attributes and methods in the diagram.

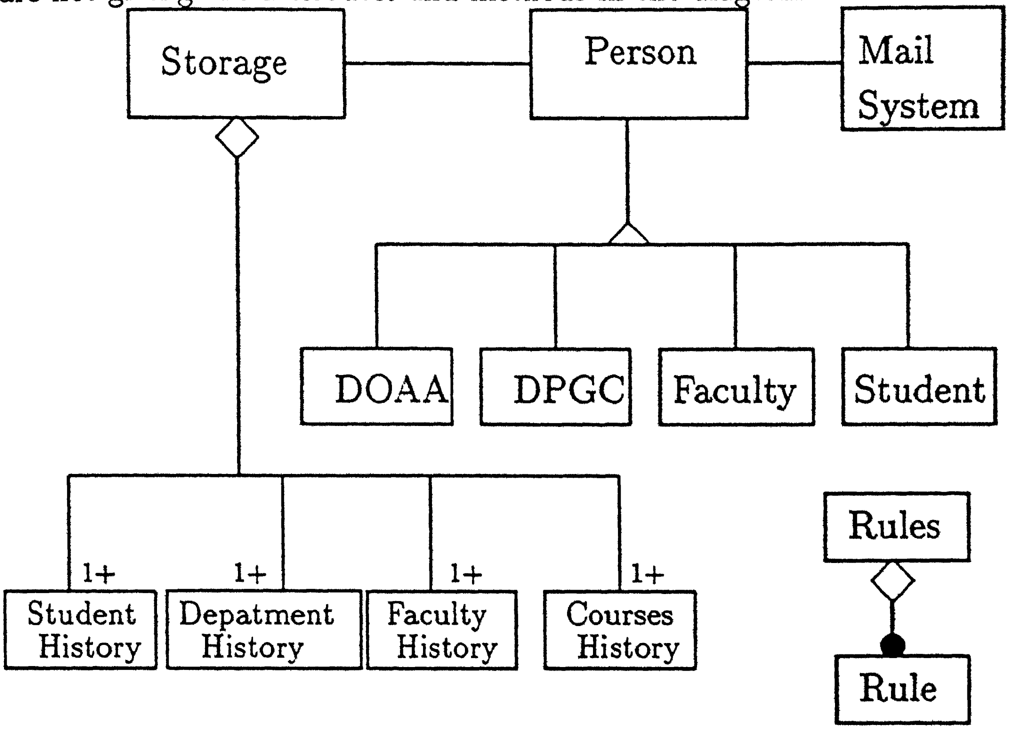


Figure 2.10 Object diagram of DOAA office automation

2.2.6 Implementation

Appendix C will give you some of the headers generated by OOMTool.

Chapter 3

User Interface

The user interface is an important part of any software component. This chapter discusses the features of the interface for OOMTool.

3.1 Overview of the OOMTool user interface

This OOMTool is implemented on "Motif". It makes use of motif widgets for its user interface. The interface of OOMTool is basically a graphical editor for OMT diagrams. The editor is divided into 5 windows: 1. Items window 2. Drawing board 3. Commands window 4. Information window 5. Help window.

Fig 3.1 shows the primary structure of the Object diagram editor. The following section(3.2) describes each part in detail. While using this editor the user will come across different sub windows through which input for different operations will be taken. Section 3.3 describes these sub windows in greater detail.

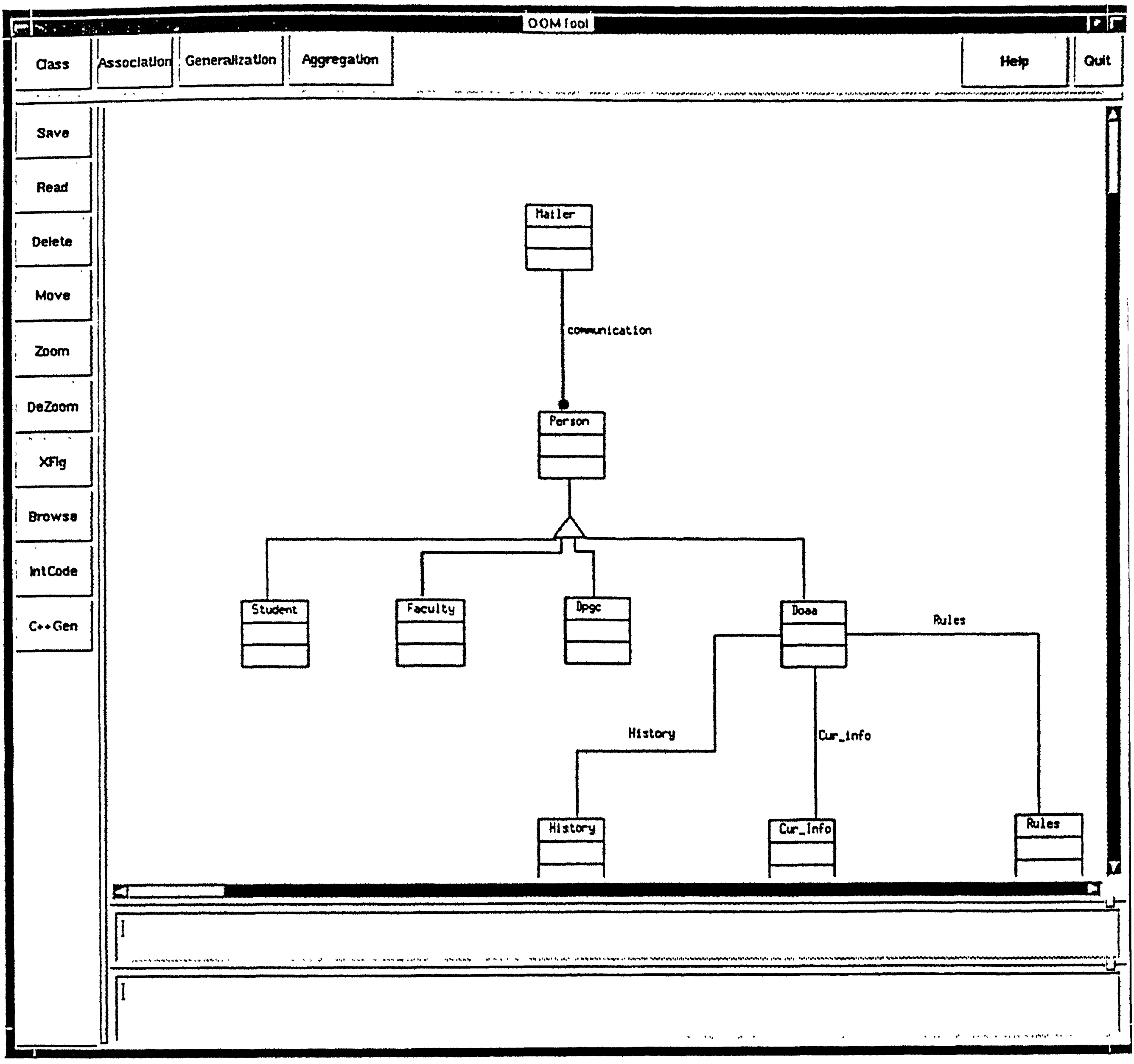


Figure 3.1 Object diagram editor

3.2 Description of Object Diagram editor

The functionality provided by each window is described in more detail in this subsection.

3.2.1 Items window

This window consists of the following 4 push buttons which are used to select the OMT item to be drawn.

- Class.
- Association.
- Generalization.
- Aggregation.

Before drawing the item, the user has to push one of these buttons. Based on the selection a subwindow is displayed to take the necessary information for drawing the selected item.

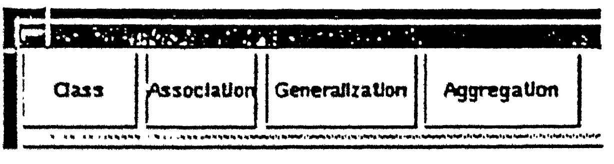


Figure 3.2 ITEMS window

3.2.2 Drawing board

This is the most important window in the editor. All diagrams are drawn in this window. There are two Scroll bars(vertical, horizontal) attached to this window. Using these scroll bars one can draw very big diagrams. As their names indicate the vertical scroll bar is used to move the diagram vertically and the horizontal scroll bar is used for horizontal movement. This window provides the user with the graphics sheet and necessary methods required to edit the diagram. These methods are:

- draw item (association, class, aggregation, generalization)
- delete item
- move item
- handle expose event
- handle resize event.
- respond to scroll bar events.

3.2.3 Commands Window

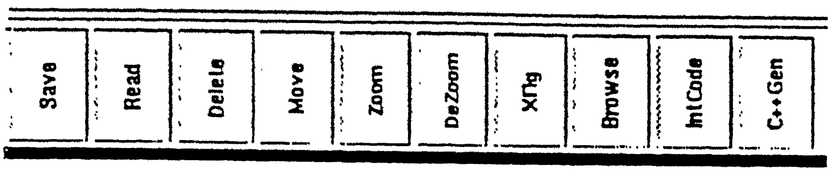


Figure 3.3 Commands window

This window consists of different command buttons which when pushed will do their intended functions. These command buttons are

- *Save*: prompts for filename and saves the diagram to the file
- *Read*: prompts for filename and reads the file into the system.
- *Browse*: waits for a click in the drawing board and if the point where the click occurs belongs to an item then shows the details of that particular item. Allows the user to modify the attributes of the items.
- *Delete*: The selected object (selection done as in browse) is deleted. While deleting consistency is checked. e.g. if the user deletes a class which is associated with another class then that association must also be deleted. Same is true for aggregation and generalization.

- *Move*: The selected object (selection done as in browse) is moved the mouse pointer until the middle button of the mouse is pressed after which the user is prompted to draw the previous relationships (associations, aggregations, generalizations) again.

- *Zoom*: shows the whole diagram in the present viewing window.

- *DeZoom*: undo the zoom function.

- *Print*: prompts for the file name and prints the drawing in XFIG format so that it can be easily converted to the required printer format.

- *CodeGen*: prompts for file name and saves generated C++ code into that file.

- *InterCode*: prompts for file name and saves generated intermediate code into the file. This intermediate code is a textual representation of the diagram and its format is present in the appendix E.

3.2.4 Information Window

There are two types of information displayed in this window:

- directions to the user while drawing different objects

- different errors in the system.

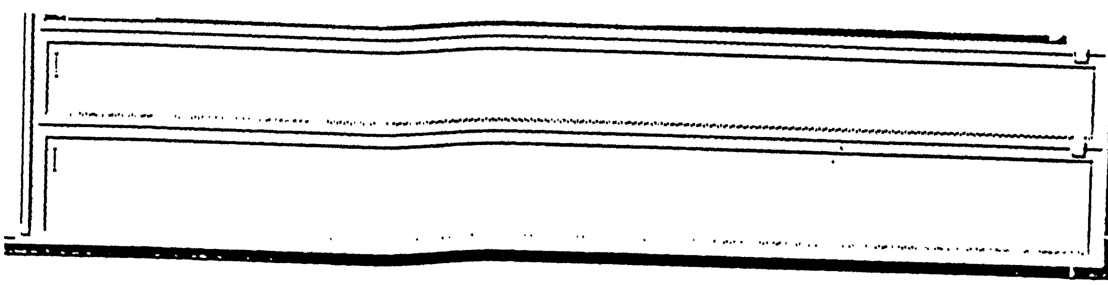


Figure 3.4 information window

3.2.5 Help Window

This window provides the user with complete online information for different aspects of OOMTool.

3.3 Description of Dynamic Diagram editor

This editor has the same structure as the Object diagram editor but with different functionality. The following section describes different subwindows displayed while using this editor.

3.4 Sub Windows

The following subsections give a pictorial view of the different sub-windows, that one may come across while working with OOMTool. Each of these sub-windows is displayed on selecting the corresponding button in the editor. The user manual describes, how to fill these sub-windows.

3.4.1 Class interface

The user has to give the Name, Attributes and Methods of a class in this window.

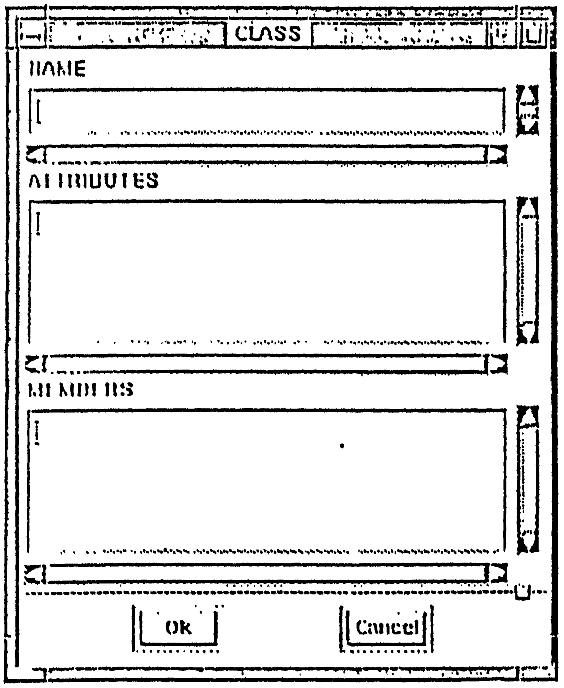


Figure 3.5 Class subwindow

3.4.2 Association interface

The Name, Multiplicities, Role names, Link attributes and qualifiers of an association are given in this following subwindow.

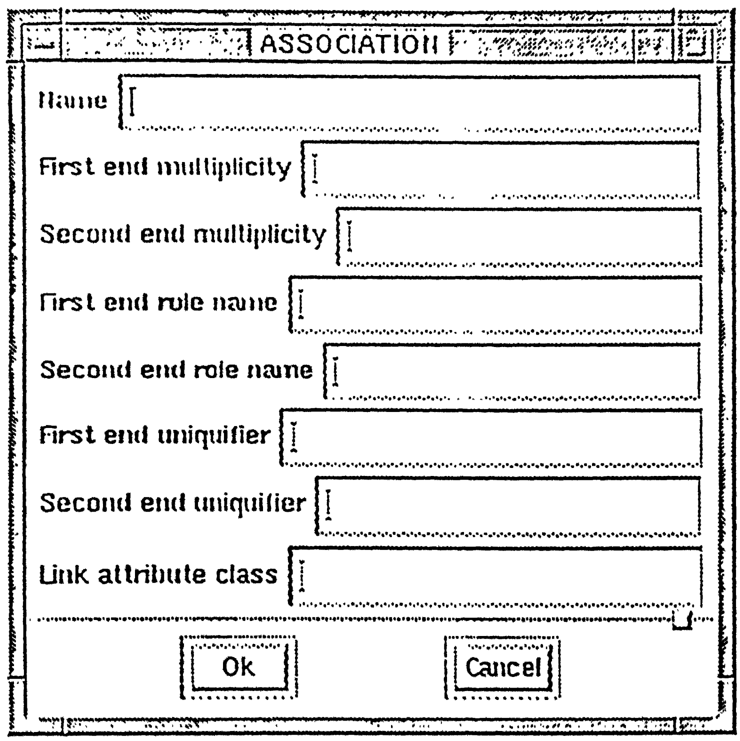


Figure 3.6 Association subwindow

3.4.3 Aggregation interface

The Name, Number of sub classes with their corresponding multiplicities are given in this subwindow.

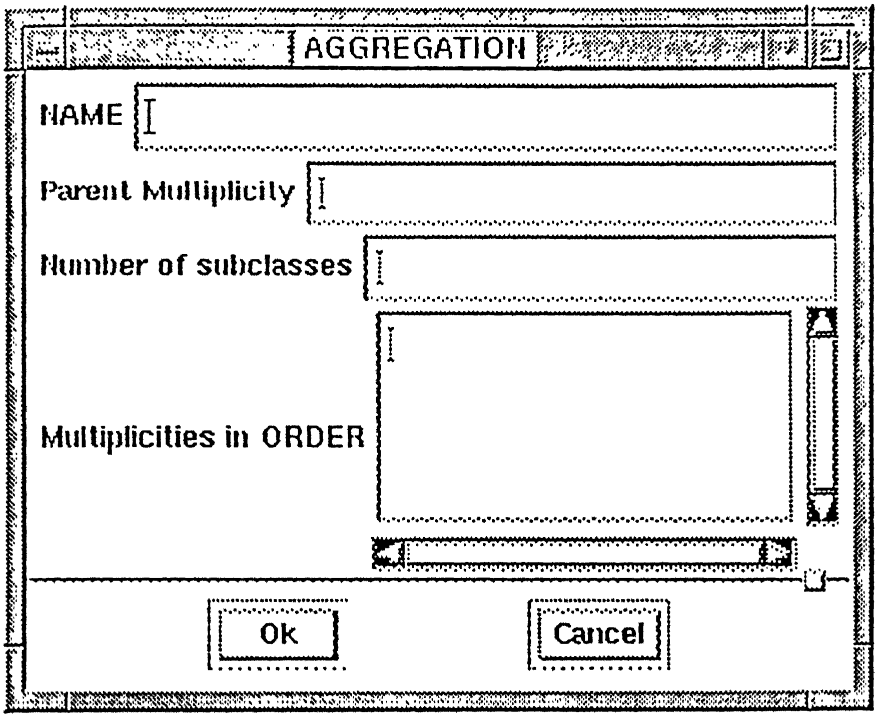


Figure 3.7 Aggregation subwindow

3.4.4 Naming interface

This subwindow is used to give the Names of different items (e.g. file names while reading, printing, saving, generalization name etc.).

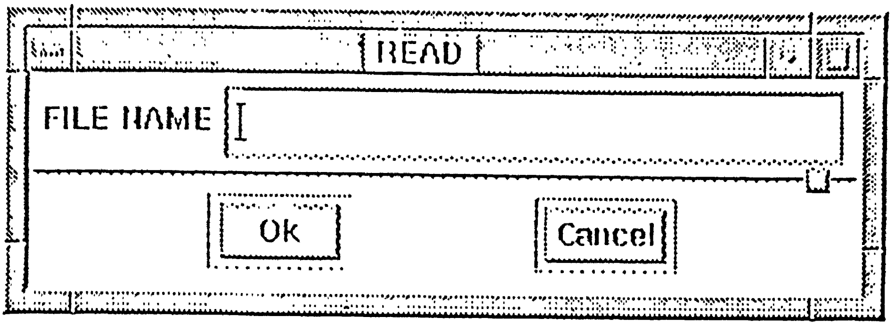


Figure 3.7 Name subwindow

3.4.5 State interface

While editing the dynamic diagram, if the "State" button is pressed, the OOMTool displays the following subwindow to take Name, Object name and type of the state. Different state types are given in the User manual.

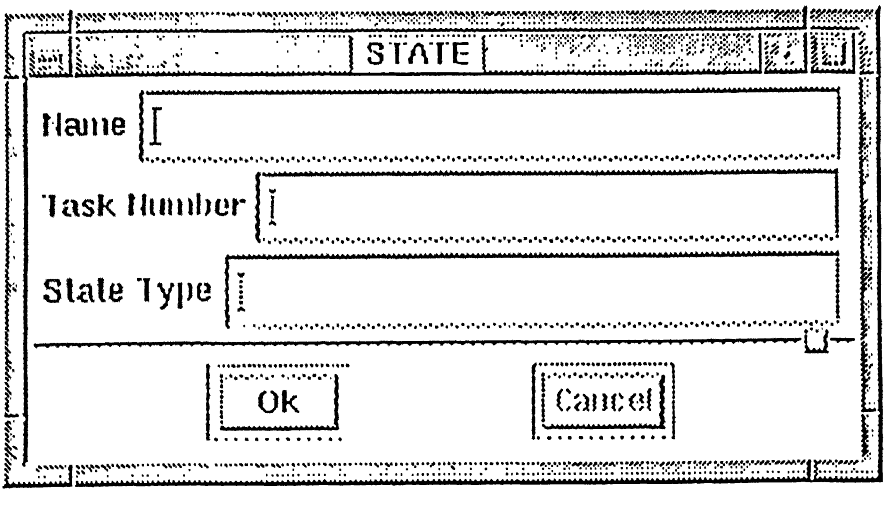


Figure 3.8 State interface subwindow

3.4.6 Event interface

While editing the dynamic diagram, if the "Event" button is pressed, the OOMTool displays the following subwindow to take Name and type of Event.

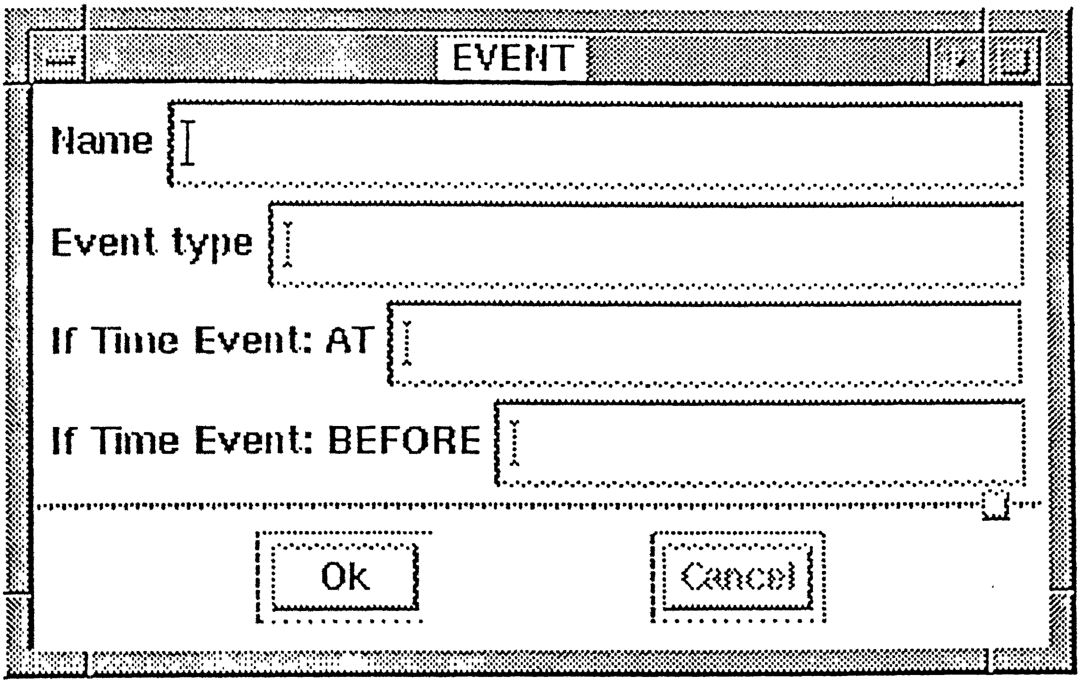


Figure 3.9 Event interface subwindow

Different event types are given in the User manual.

Chapter 4

Development of OOMTool

4.1 Problem Statement

Develop a system which can help the user to develop the software using Rumbaugh's Object Modeling Technique. This system must provide support to edit OMT Object and Dynamic diagrams, and should also produce as much C++ code as possible using these two diagrams.

4.2 Requirements Analysis

From the above problem statement, the following 3 major modules are identified.

1. Editor for Object diagram
2. Editor for Dynamic diagram
3. Code generation

4.3 Design

This system has to perform similar functions (e.g. drawing, deleting, moving, storing, reading, etc.) on different types of objects (eg: classes, associations, aggregations, states, events etc.). The attributes and functions of each object are well defined. We use the C++ language and X-Windows to implement OOMTool.

The following sections develop the 3 modules mentioned in the requirements analysis section.

4.3.1 Development of the Object diagram Editor

Requirements Analysis: (Problem Statement)

This editor should provide the following

1. Very large drawing area: in which all diagrams can be drawn
2. Support for drawing different symbols of OMT
3. Deleting selected symbols
4. Moving selected symbols
5. Support for scroll bars which can help in viewing a selected area in the drawing.
6. Intermediate modifications to the different items.
7. Zooming and dezooming.
8. Directions to the user while drawing
9. Reporting errors with possible explanation
10. Help.

Identification of Objects

The following objects are identified from the above problem statement.

1. drawing area and scroll bars
2. icons for different symbols in the OMT Object diagram.
   - Class
   - Association
   - Aggregation
   - Generalization
3. icons for different commands for editing the diagram
   - Save
   - Read
   - Delete
   - Move
   - Browse
   - Zoom
   - DeZoom
   - Print etc.
4. Storage: buffer for the diagram being edited.
5. Message area: all messages are displayed here
6. Help

Identifying the relationship between objects.

1. Drawing area, scroll bars, and the buffer used to store the current diagram are tightly related to each other. We create a class called drawing board as an *aggregation* of the above mentioned three classes.

2. The buffer has to store different OMT symbols. At any particular instant of time this buffer is an *aggregation* of zero or more objects where each is one of the OMT symbols. So buffer is a *generalization* of different OMT symbols: class, association, aggregation, generalization etc.

3. Items Window: As the icons for Class, Association, Aggregation, Generalization have similar functionality, we group them(*aggregate*) together in ItemsWindow.

4. Command Window: All the commands are grouped together in the CommandsWindow.

5. Editor object: This object is an aggregation of the 4 objects(in 1 to 4). This will maintain the state of the editor.

6. Message Window: As this window is being used by all the classes in the system, we keep it as a global object.

Dynamic nature of the Editor object.

At any instant of time, depending on user interaction the Editor object may be in any one of the following states.

- Initial state: No work is being done.
- Input State: Subwindow is displayed and waiting for input corresponding to the user selection
- Item Drawing : one of the items is being drawn.
- Command Processing: one of the commands is being processed.

Details of the events and some other states are shown in fig 4.2

Object diagram

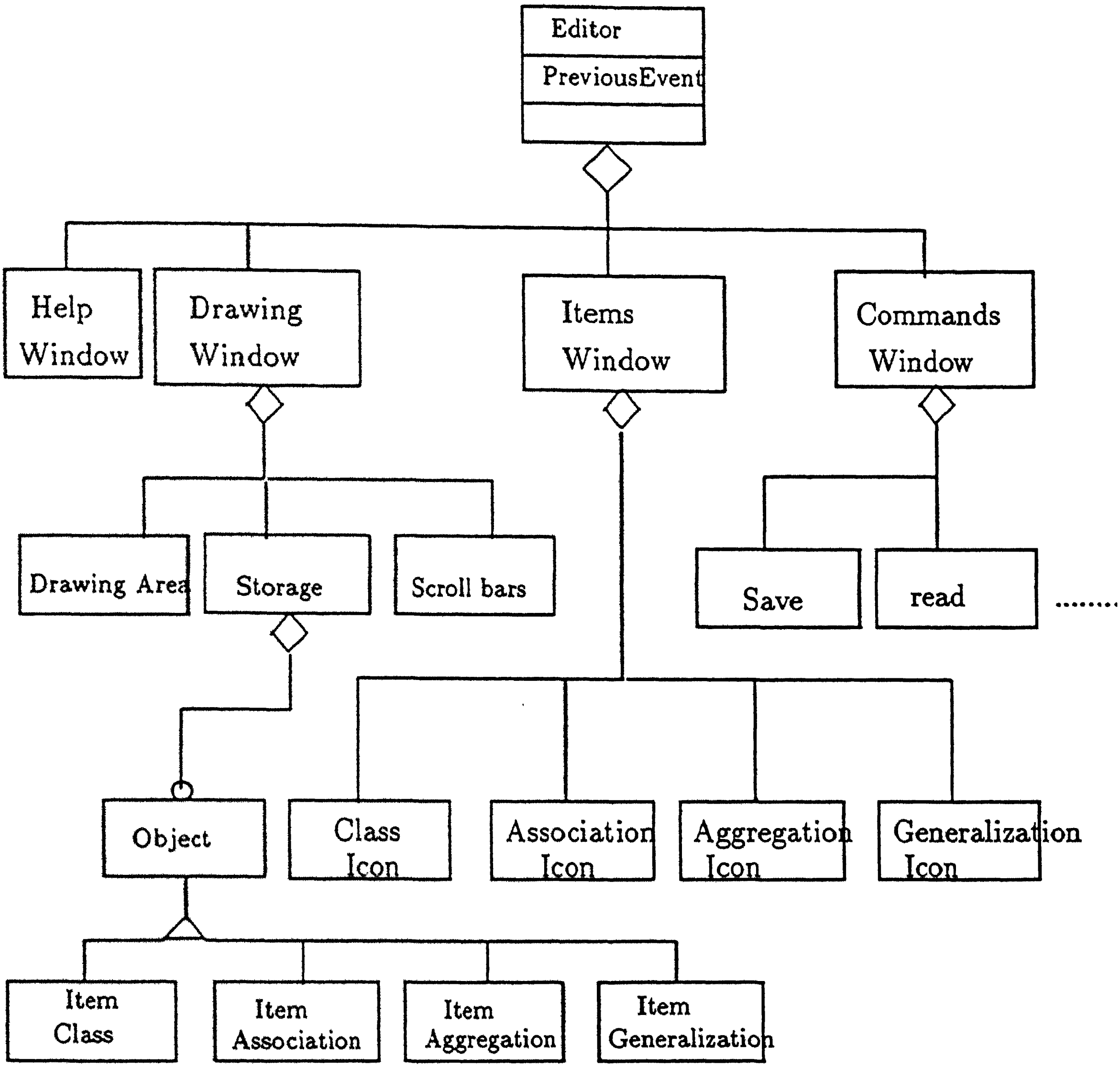


Figure 4.1 Object diagram for Object diagram editor

Dynamic diagram

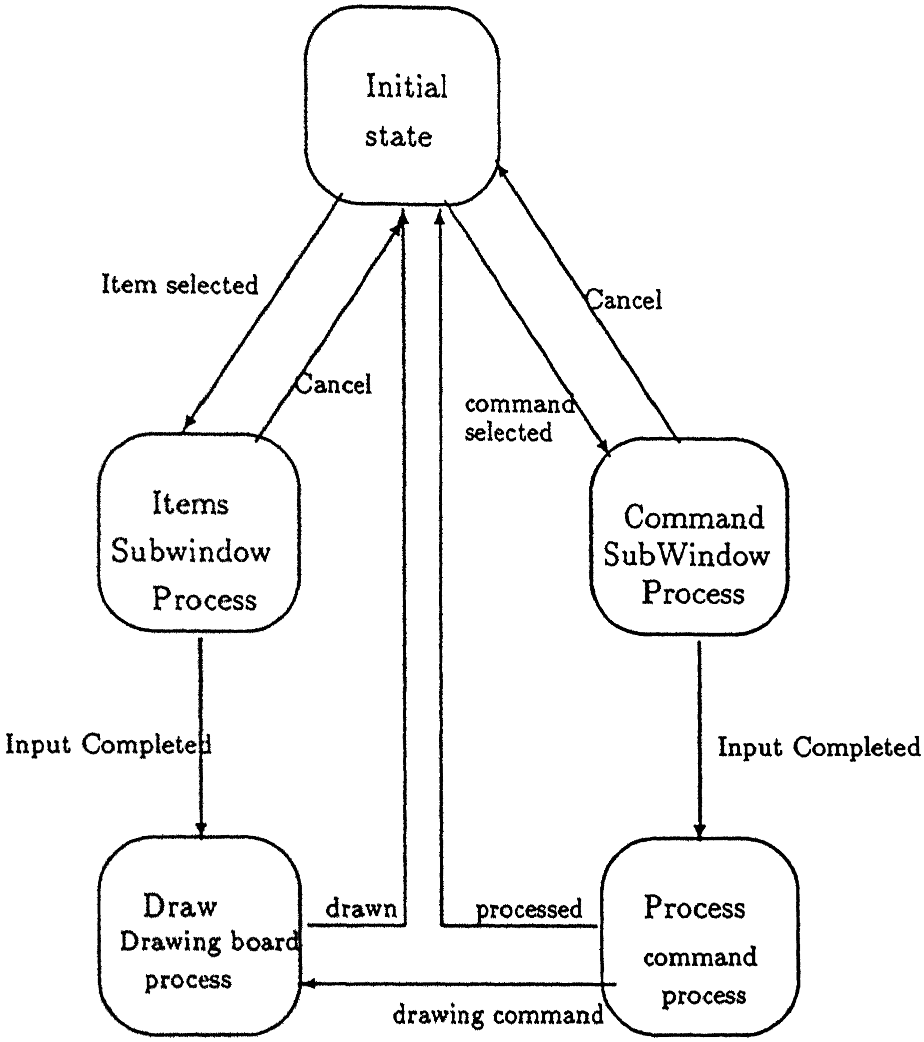


Fig 4.2 Dynamic diagram for object Editor

Implementation

Appendix B gives some of the C++ class headers required to implement the OOMTool.

4.3.2 Development of the Dynamic diagram Editor

This is similar to Object diagram editor but uses different symbols. Extra features provided in this editor are:

1. State generalization: Nesting of the states is supported i.e allow the user to iconify some states and events of a dynamic diagram.

2. Iconification of whole dynamic diagram of a particular object. During this, a dynamic diagram of a selected object will be replaced by an icon with places for input states and output states (see figA.3). All external events (events from one object to other object) can be drawn only after iconifying the dynamic diagrams of the two objects.

4.3.3 Development of the Code generation module

Requirements analysis

1. Generate intermediate code for the drawing: drawing consists of different symbols in OMT, linked together.

2. From the generated intermediate code, generate C++ code.

3. Generated code must maintain different relationships between objects (e.g. association multiplicity and referential integrity etc.)

Design

Intermediate code generation : By using the diagram in textual form, one can easily write the code generation modules for the required target language. The following section explains the code generation module for C++. Store the minimum information in textual form so that all the information can be retrieved without any loss. Format of this intermediate code is given in the appendix E.
C++ Code Generation: Mapping OMT symbols to the C++ code is the important thing while designing this module.

Some of the OMT elements like *Class* and *Generalization* have a direct C++ mapping. Eg: OMT *Class* can be mapped to C++ *class* and OMT

Generalization corresponds to C++ *Inheritance*. But *Association* and *Aggregation* have no direct mapping in C++. This is where we have to map them in such a way that all the conditions are satisfied. The rest of this section discusses these mappings in greater detail.

- *Class*: OMT Class can be directly mapped to C++ class as follows: Attributes of OMT class will be the data variables of C++ class and operations(services) of OMT class will be mapped to methods of C++ class.

  Default accessibility: attributes are private and methods are public.

  Default type: attributes are void type and methods are void* type.

  If User mentions the accessibility of the attributes and methods those things will be preserved.

  C++ code for figure 2.1 will be:

```
class ClassName {
    private: // default access for data vars
        dataType1 attribute1, attribute2;
        dataType2 attribute3, attribute4;
    public: // default access for methods
        resultType operationName1(argList);
        resultType operationName2(argList);
}
```

- *Association*: Our mapping of *Association* into C++ must maintain
  - Multiplicity: At any instant of time a valid object must have the relationship with specified number of other objects.
  - Qualifiers: Associated objects must be distinguishable with respect to this qualifier. i.e no two associated objects will have the same value for this qualifier
  - Link attribute: is a property of the link in an association.

- Referential integrity: is the mechanism to ensure that if an object x of class X is having a pointer to an object y of class Y, then there must exist a pointer to object x in object y.

Basically, this is mapped into C++ by having each class contain a pointer to the other participant. But a problem with these pointers is that the above four requirements are not met. To maintain these requirements we are using C++ template based mechanism.

Template based mechanism: in this method, instead of a pointer to the other participant we will have an object of this template class. Eg

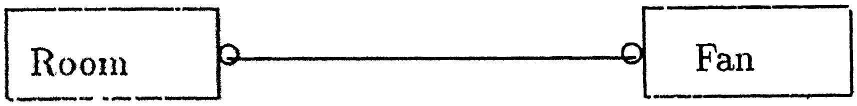


Figure 4.3 A simple association

Code for above figure will be:

```
class Room {
        private:
          Mult<Fan> FanPtr;
        }


class Fan {
        private:
          Mult<Room> RoomPtr;
        }
```

This Mult<> template has the following features:

- Maintain Multiplicity: if multiplicity is *one* and if any object is being associated then it will replace the previous object. If multiplicity is *many* and if any object is being associated then it will add that object to existing object set. It will do the same for other multiplicities also.
- Provides a function to check the validity of the link.

- Maintain referential integrity: provides two functions *link* and *unlink* which can insert/delete the pointer of other participant. While doing this insertion/deletion it also inserts/deletes the corresponding pointer in the other participant.

Relating to this template instantiation OOMTool will also generate two functions in the class where this instantiation is being done. These two functions are also called *link* and *unlink* which inturn call the corresponding link and unlink functions of template. If two or more associations exist with the same class then these functions will be overloaded.

The object of any class is valid only if all the requirements are satisfied. So to check this validity OOMTool generates one more function called checkIntegrity() which can check the validity of the object. One can see the appendix C for some generated class headers for the example designed in section 2.2.

Now we will show actual implementation of different attributes of an object.

- Multiplicity: tells the number of objects associated with it. This multiplicity may be
  * Optional: associated with zero or one objects. At any particular instant of time a pointer may have at most one value or NULL. So the template class will have only one pointer and checkValidity function always returns true.
    Eg: see Figure 4.3
    Code:

```
class Room {
    private:
        OptMult<Room,Fan> FanPtr;
    }

class Fan {
    private:
        OptMult<Fan,Room> RoomPtr;
    }
```

* many: associated with zero or more objects. We can map this to an array of pointers. So the template class will have an array of pointers and checkValidity function always returns true.

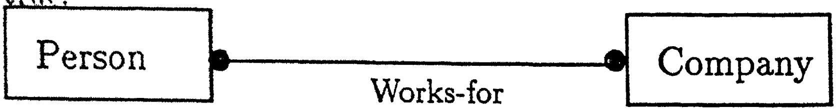


Fig 4.4 simple association

Code for above figure will be:

```
class Person {
      private:
         ManyMult<Person,Company> CompanyPtr;
      }

class Company {
      private:
         ManyMult<Company,Person> PersonPtr;
      }
```

* one: associated with exactly one object.

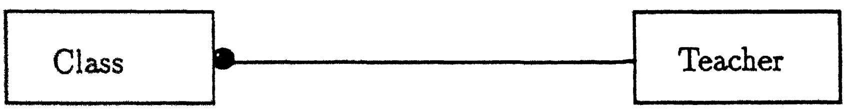


Fig 4.5 example for Multiplicity ONE.

The object of class "Class" will have the meaning only when it is associated with exactly one object of class "Teacher". To help the user maintain this constraint we have to generate the code in such a way that an error or an exception is raised whenever some of the attributes or methods are accessed with out satisfying this constraint. For this purpose our template will have a pointer and checkValidity function returns True if one object is associated and returns False otherwise.

Code for above figure will be:

```
class Class {
      private:
         OneMult<Class,Teacher> TeacherPtr;
```

```
        }

class Teacher {
        private:
           ManyMult<Teacher,Class> ClassPtr;
        }
```

* range(n+): associated with "n" or more number of objects. We can map this to an array of pointers. So the template class will have an array of pointers and checkValidity function returns True if at least "n" objects are associated and False otherwise.

Figure 4.5 range Multiplicity

Code for the above figure will be:

```
class Line {
        private:
           rangeMult<Line,Point,2> PointPtr;
        }
class Point {
        private:
           OptMult<Point,Line> LinePtr;
        }
```

* set: (eg: (2,5,7)): associated with 2 or 5 or 7 number of objects.
  Eg: This is also same as above, but instead of rangeMult<LhsClass, RhsClass,n> here we will have setMult<LhsClass,RhsClass,set>.

– Role Names: will tell the roll of the other end object with respect to this object. This attribute has nothing to do with the code and this is used to make the design description more clear. We can use this as the name of the pointer. As an example the code for figure 2.5 is shown below.

```
class Person {
```

```
private:
   OneMult<Person,Company> employerPtr;
}
```

```
class Company {
      private:
         ManyMult<Company,Person> employeePtr;
      }
```

- Qualifiers:

  For an example, see the figure 2.3. This figure shows that at any particular instant of time, a directory will be associated with the files with unique file names i.e no two files have the same name. This is applicable only to many-to-many or one-to-many type of associations. To maintain this uniqueness we can write a template. Code for the figure 2.3 is

  ```
  class Directory {
        private:
           UniqMult<Directory,File,FileName> FilePtr;
        }
  ```

  ```
  class File {
        private:
           OneMult<File,Directory> DirectoryPtr;
        }
  ```

  Whenever object of type File is being inserted in the directory object, this template will make sure that the name is unique.

- Link Attributes:

  For an example see figure 2.4. We can model this link attribute itself as a class and we reorganize the above figure as follows.

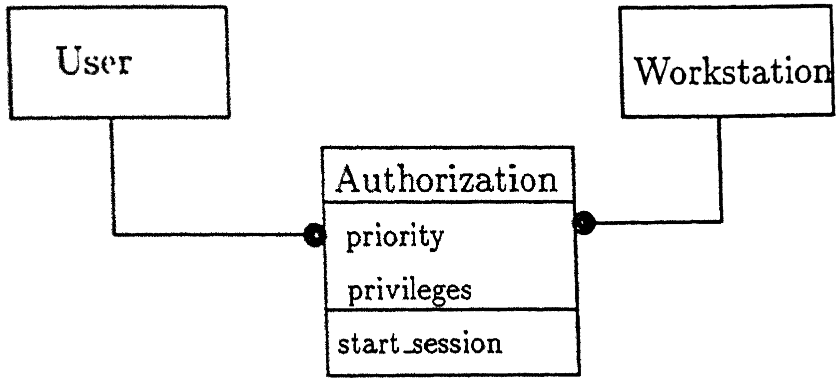


Figure 4.7 Link attribute and Modelling an association as a class

Now we can translate it as per our above design. In the link attribute class templates are instantiated as public members instead of private, to enable the classes in association to refer each other.

Some of these template definitions are available in appendix D.

- Aggregation is a part-whole or a part-of relationship. In this relationship the components are associated with an object representing the entire assembly. We can model this as a one-way association. Multiplicity will be maintained as per the rules mentioned for association multiplicity. As it is a one way association OOMTool provides some other set of template classes for aggregation multiplicity, to reduce the complexity. For example see figure 2.6. Code for this figure is

```
class Document {
    private:
        AgManyMult<Paragraph> ParagraphPtr;
    }

class Paragraph {
    private:
        AgManyMult<Sentence> SentencePtr;
    }
```

```
class Sentence {          }
```

- Generalization This concept is related to *Inheritance* in C++. We can generate the code as follows. For example code for the DOAA object in the figure 2.7 will be

```
class DOAA : Person {
        }
```

Conclusions and Further Work

In the previous chapters we have discussed the design and implementation of OOMTool. This system provides the users with editors for Object and Dynamic models and a facility to generate reliable C++ code. Using these editors and code generation module one can easily translate the design into implementation.

If any changes are there in the design, perceiving those changes and writing code accordingly gets complicated with the increase in number of objects for a large system. This OOMTool can be used to overcome this problem and it will also enhance the productivity and accuracy of the system developer.

Further Work

1. Editor: A module which can adjust the drawing automatically while the current diagram is being modified can be developed. e.g. if the class symbol is moved from one place to another, it is necessary to redraw the previous relationships(associations, aggregations etc.). At present ths has to be done manually.

2. Intermediate Code:As the intermediate form of the diagram is available, more code generation modules can be added for other languages or database systems. At present C++ code is being generated

3. Meta CASE:Some interface can be provided to the user so that he/she can

    - define new symbols in the diagrams
    - specify how code should be generated for different symbols

References

[1]. Object Oriented Modeling and Design by James Rumbaugh, Michael Blaha, William premerlani, Frederic Eddy and William Larenser. 1991, Prentice-Hall,Inc.

[2]. Object-Oriented Analysis by Peter Coad/Edward Yourdon. 1990, Object international,Inc.

[3]. Object Oriented design and applications by Grady Booch. Publisher: Benjmin Cummins, 1994

[4]. An Introduction to Object Oriented Programming by Timmothy Budd. 1991, Addesen Wesley.

[5]. A Practical Strategy for Object Oriented Design by Kanchan Kumar. Dr.Dobb's Journal, June 1995.

[6]. Comparing CASE Tools by Jeffrey L. Armbruster. Dr.Dobb's Journal, June 1995.

[7]. The Object Oriented Model and Its Advantages
by Jose de Oliveire Guimaraes.

[8]. Single Versus Multiple Inheritance in Object Oriented
Programming. by Ghan Bir Singh.

[9]. The Essence of Objects: Concepts and Terms
by Alan Snyder. IEEE Software, Jan 1993.

[10]. A proposed standard set of Principles for Object Oriented
Development. by David C.Rine. ACM SIGSOFT Software
Engineering Notes vol16 no 1, Jan 1991.

[11].Motif Programmer's Manual, Volume 6 from O'Reilly &
Associations.

[14].The C++ Programming language, second edition,
by Bjarne Stroustrup, Addisen wesely 1994.

Appendix A

User's Manual

A.1 Hardware and Software Requirements

OOMTool runs on any machine which supports X windows. It requires Motif library to be present with the X windows. OOMTool is implemented in GNU C++ version 2.7.0

A.2 Starting OOMTool

To start OOMTool type :

$ OOMTool <CR>

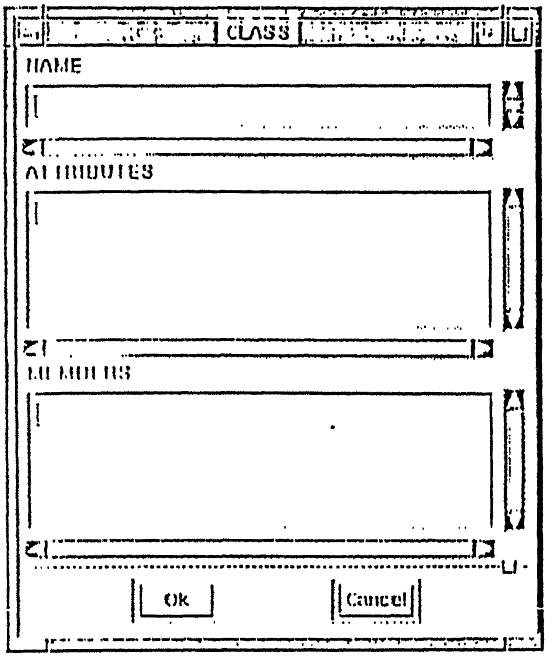
It will display a screen through which user can interact with the system. As a default editor it gives Object Diagram Editor. You can switch to Dynamic diagram editor by pressing the icon called DDEditor in the command area.

A.3 Object Diagram Editor

Here we will illustrate usage of various functions of the editor.

A.3.1 Drawing Class symbol

- Push the pushButton named "Class" in the Item Menu.
- It will display the following subwindow to take the required inputs



  - give Name of the class in Name area
  - give Attributes with data types and access specifications(public, private, protect etc.)in the "Attribute area".

    Syntax:

    ```
    [ [Access:]
       [dataType1]: Attr1,Attr2,.....;
       [dataType2]: Attr3,Attr4,.....;
       ..............
    ]*
    ```

    Default Access: private

    Default dataType: void
  - give methods with result types and access specifications in the "Methods area"

    Syntax:

    ```
    [ [Access:]

       [resultType1] method1(arg1,arg2,....);
    ```

[resultType2] method2(arg1,arg2,....);

....

]*

Default Access: public

Default resultType: void*

After filling all the above details press OK button to draw the Class diagram or Cancel button to cancel.

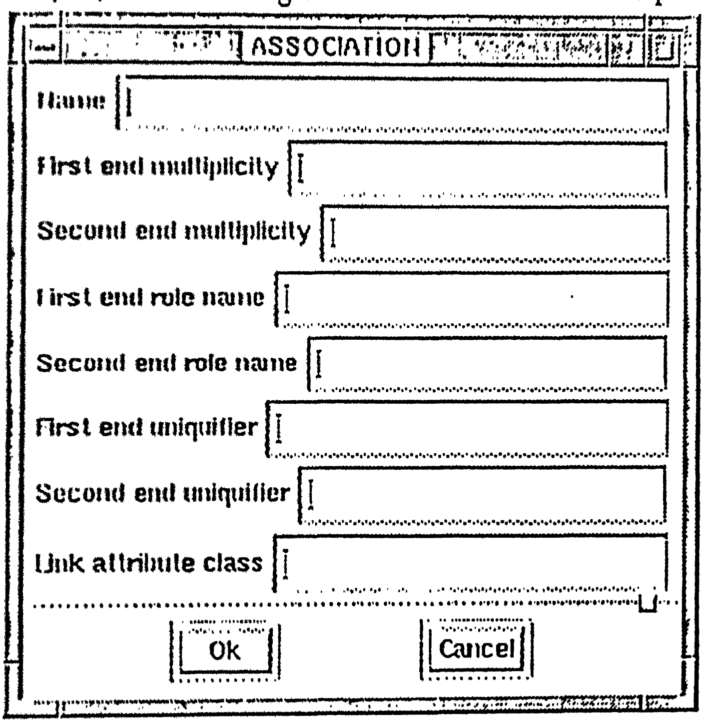
- If you press cancel, subwindow disappears and you can start an other operation.

- If you press OK button, subwindow disappears and the drawingBoard is ready to draw the class diagram. By moving the mouse, you can select the place where you want to draw the class diagram. By pressing the mouse first(left)button you can see the class diagram on the drawing board. You can move this diagram using Move pushButton. We will illustrate this Move button in the following section.

A.3.2 Drawing Associations

- Push the pushButton named "Association" in the Item Menu.

- It will display the following subwindow to take the required inputs.



You can fill the required information and leave unnecessary fields.

- give the name of the association
- give multiplicities at the two ends
  allowed multiplicities: opt, one, many, n+(n is any number), (m,n,...)where m,n are digits
  Default: optional(opt)
- give roleNames. Default:NULL
- give qualifiers. Default:NULL
- give link attribute class Name. To draw the link attribute you should have drawn the link attribute class previously. Default:NULL

use OK and Cancel buttons as above.

- If you press OK button, all the other buttons will be disabled and the drawingBoard is ready to draw the Association. Move the mouse nearer to the class at which association starts and press first(left) button. Now
  - if you move the mouse while pressing left button you can see the rubber band line.
  - if you press third(right) button, the line drawn up to that point will be fixed and the rubber band line starts again from that point.
  - if you press middle button, if that point belongs to any one of the classes in the drawing then line drawing will be stopped.
  - press Escape to cancel the drawing.
  - use h,j,k,l keys to scrol the drawing board while drawing.
    h− >left, j− >down, k− >up, l− >right

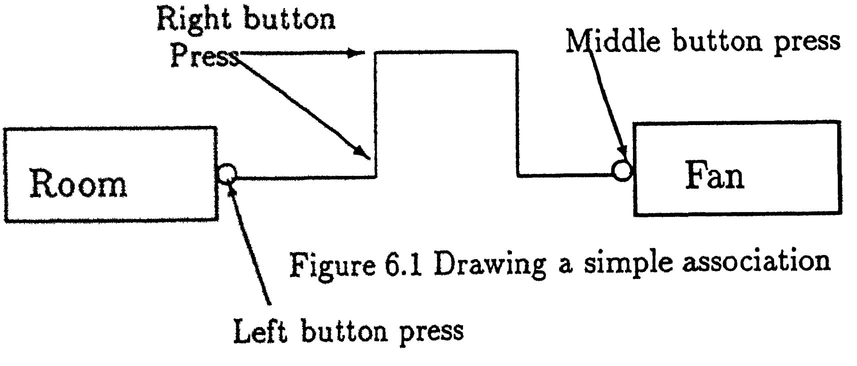


Figure 6.1 Drawing a simple association

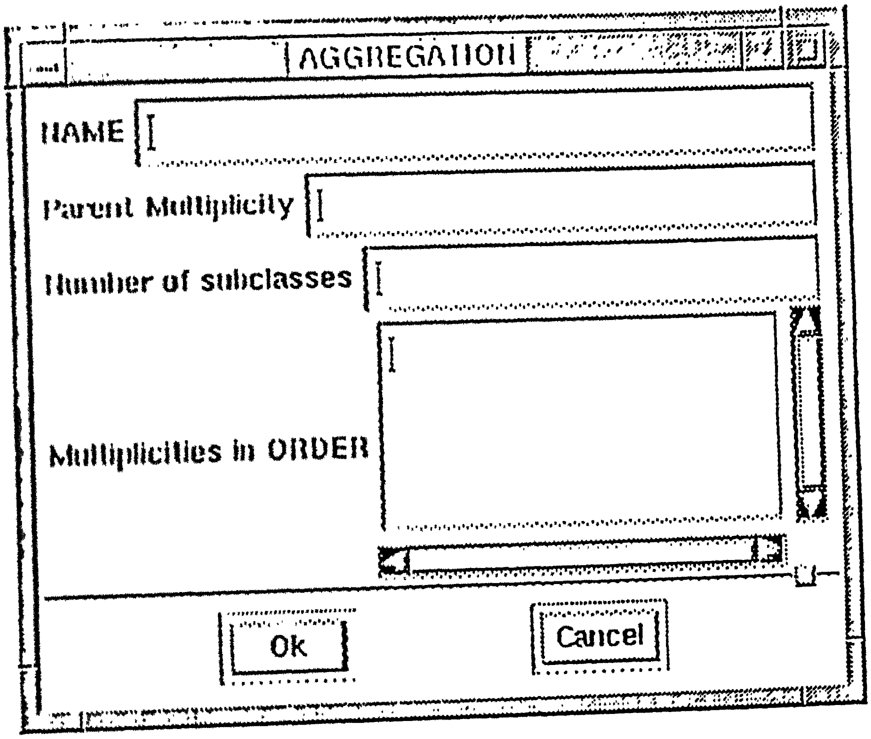
After completing the line drawing, system will prompt for places to draw the roleNames, multiplicities etc. You can use left button and middle button to draw these things.

If link attribute is not Null, now system is ready to draw the link of the link attribute. To draw this, you have to give three points in the follwing order: 1.point on the association line, 2.point on the boundary of link attribute class, 3.point on the association line. You can give these points by moving and pressing the left button of the mouse.

- While drawing, directions will be displayed in the message area of the editor.

A.3.3 Drawing Aggregation

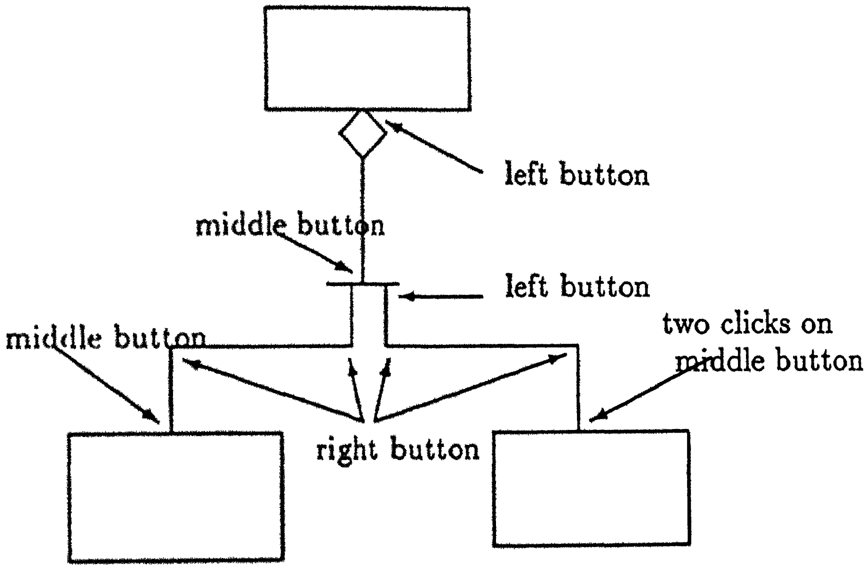
- Push the pushButton named "Aggregation" in the Item Menu.
- It will display the following subwindow to take the required inputs



  - give the name of aggregation
  - give the number of subclasses
  - give the multiplicities of the subclasses in order.
    Syntax : same as association multiplicity

use OK and Cancel buttons as above.

- If you press OK button, all the other buttons will be disabled and the drawingBoard is ready to draw the Aggregation. Move the mouse nearer to the container class at which aggregation starts and press first(left) button. Now you can see the aggregation symbol. Using the first button only you can draw rubberband line. Push middle button at the point where you want to start the splitting(see fig). From this split point you can draw each branch with the procedure used to draw the association. At the last branch end you have to double click the middle button.



A.3.4 Drawing Generalization

- Push the pushButton named "Generalization" in the Item Menu.

- It will display a subwindow to take the Name of the Generalization.

  You can draw the generalization using the same procedure used to draw aggregation. Aggregation split point corresponds to the triangle symbol in the generalization.

A.3.5 Usage of the Command buttons

This subsection illustrates usage of various command buttons in the command area.

- Save the diagram to a file:
  - Push the pushButton named "Save" in the Command Menu .
  - It will display a subwindow to take the Name of the file. It saves the drawing to the specified file
- Reading from a file:
  - to read an already saved file into the drawing, push pushbutton "Read"
  - It will display a subwindow to take the Name of the file. It reads the file into memory.
- Deleting some symbol from the drawing:
  - Push the pushButton named "Delete" in the Command Menu.
  - Now the drawing board is ready to delete one of the symbols. Move the mouse and click left button on the symbol you want to delete. If it finds any symbol, it will delete that symbol.
  - while deleting the object, system will check for consistency eg: If you delete a class and if that class is associated with some other class then that association will also be deleted. This will apply for aggregation and generalization also.
- Moving some symbol in the drawing:
  - You can move only class symbols.
  - Push the pushButton named "Move" in the Command Menu .
  - Now the drawing board is ready to move one of the class symbols. Move the mouse and click left button on the class symbol, that you want to delete. If it finds any class, it will remove that class and its relations from that place and wait until the middle button s pressed. If you press middle button it will draw the class there and prompts the user to draw all the previous relations.

- viewing total drawing at a time: Zoom
  - Push the pushButton named "Zoom' in the Command Menu.
  - system will show total figure in the present viewing window.
- DeZoom: it will undo the previous command
- Print intermediate information
  - Push the pushButton named "InterCd" in the Command Menu.
  - It will display a subwindow to take the Name of the file. It prints the intermediate code of drawing to the specified file
- Print C++ code(CODE GENERATION)
  - Push the pushButton named "CodeGen" in the Command Menu.
  - It will display a subwindow to take the Name of the file. It prints the C++ code to the specified file
- Dynamic diagram editor:
  - Push the pushButton named "DDEditor" in the Command Menu.
  - Now You can see the Dynamic diagram editor

A.4 Dynamic diagram Editor

A.4.1 Editor

In this editor drawing "states" corresponds to drawing "classes" in the Object diagram editor,and drawing "events" corresponds to drawing "associations".

While drawing the states or events subwindows to take the inputs, will be displayed. Give the inputs here and draw them.

States are of four types:

- input state : external events can come to this state (type:i)
- output state : external events can start from this state(type:o)

- input and output state : external events can start from this state and come to this state(type:io)
- internal state: no external events(type:internal)

Events are of two types:

- internal events: events between two states of the same object (type:interanl)
- external events: events between two states of the different object(type:external)

Extra facility provided here is Iconification of whole dynamic diagram of one object. You can draw the external events only after Iconification. After Iconification you will see the dynamic diagram as follows.

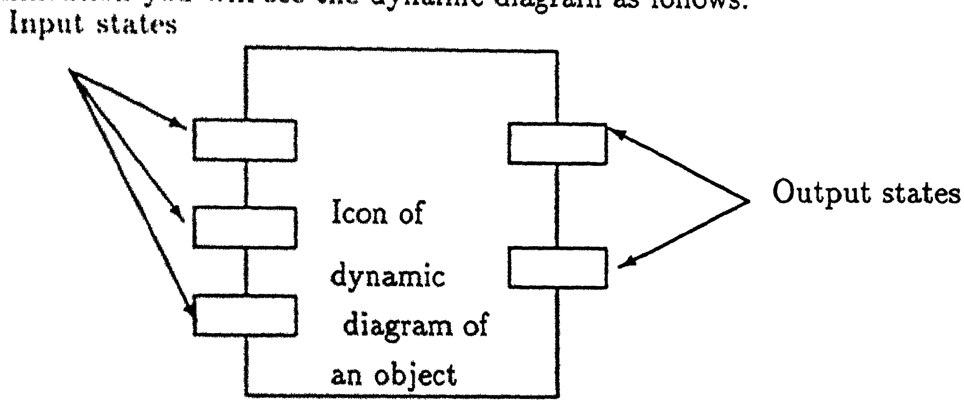


Figure 6.3 Icon for the dynamic diagram

In the above icon, left hand side boxes corresponds to input states of the Object and right hand side boxes corresponds to output states.

Appendix B

OOMTool implementation

This appendix gives the C++ headers required for the OOMTool implementation. We are giving a very brief and vital parts of the headers.

```
//Editor class is an aggregation of DrawWindw, ItemWindow,
// CommandWindow, and HelpWindow classes.
 class Editor{
  private:
   DrawWindow    drawWindow;
   ItemWindow    itemWindow;
   CommandWindow commandWindow;
   HelpWindow    helpWindow;
   int           preEvent; //stores the user's previous selection
                           //using this drawing window acts
  pubic:
  Editor(){
   preEvent = -1; //no selection
   }
```

```
  void editorSetUp(){
    // create main window
    // Ask the 4 windows to create their corresponding widgets
       in this main window and add corresponding call backs.
    }
};


//DrawWindow class is an aggregation of DrawArea, Storage, ScrolBars
// This class coordinates the functionality of these above three classes.

 class DrawWindow{
   private:
    DrawArea drawArea;
    Storage  buffer;
    ScrolBars scrolBars;

   public:

    void handleScrolBarEvents(){
    //On receiving the events, Scrol bars update their positions
     and ask the drawing window to update the drawing.
    //This function asks the drawArea to scrol the diagram and it
      supplies the buffer to the drawArea
    }

    void handleDrawAreaEvents(){
     //After completing the current symbol drawing drawArea asks the
```

```
      drawWindow to store the current figure.
      // This function asks the buffer to store the current figure.
      }
};

// Storage class is an aggregation of Object classes where each
// Object is one of the OMT symbols i.e Object is generalizaion of
// OMT symbols (Class, association, aggregation, generalization).

class Storage{
   private:
     Object **objects;
     int    noOfObjs;


   public:
     Storage(){
       noOfObjs = 0;
       }

     void readFile(char *fileName); // read the diagram from the file
     void saveFile(char *fileName); // save the diagram into the file
     void intermediateCodeGen(char *fileName); //save non graphics info
     void insertObject();//insert given OMT object
     void deleteObject();//delete given OMT object
     void printXfig(char *fileName); //convert the drawing into Xfig
                                     // format and print
```

```
};

class DrawArea{
   private:
     widget drawingBoard;
     int    status; //normal or zoomed

   public:
     DrawArea{
      status = 1; //normal
      }
     void createDrawBoard();//create drawingBoard widget and add call bacs
     void ReDraw();
     void Scroll();
     void interactiveClassDraw();
     void interactiveAssociationDraw();
     void interactiveAggregationDraw();
     void interactiveGeneralizationDraw();
     void handleExposeEvent(); //Redraw whole diagram
     }


// ItemWindow class is a container class i.e it has no direct
// functionality It will manage all of its parts(ClassIcon,
// AssociationIcon, AggregationIcon, GeneralizationIcon
```

```
class ClassIcon{
    private:
     widget classIcon;

    public:
     void createClassIcon();
     void displayInterface(); // interface to take inputs
                              // e.g. attributes, methods, Name etc.
    }

// AsscioationIcon, AggregationIcon, GeneralizationIcon classes are
// similar to the above ClassIcon


// As the ItemWindow class Command window class is also a container
// class i.e it has no direct functionality. It will manage all of
// its parts(saveIcon, readIcon, printIcon, browseIcon etc.

class SaveIcon{
   private:
    widget saveIcon;

   public:
    void createSaveIcon();
    void displayInterface(); // interface to take fileName
    void sendEventToDrawWindow(); //ask to save the diagram
    };
```

```
// Other command classes will also diaplay some inerface and ask the
//   drawWindow to carry on the function
```

```
      int checkIntegrity(){
        PersonPtr.checkIntegrity();
      }



      void SendMail() {
         checkIntegrity();
         PrivateSendMail();
         }

      void ReceiveMail() {
         checkIntegrity();
         PrivateReceiveMail();
         }

      inline int link(Person *PersonObjPtr, int needBackTrace){
       return(PersonPtr.link(this,PersonObjPtr,needBackTrace));
      }
      inline int unlink(Person *PersonObjPtr){
       return(PersonPtr.unlink(this,PersonObjPtr,needBackTrace));
      }
};

class Person{
    private:
      char *Name ;
      char *Address ;
```

```
    char Sex ;
    OneMult<Person,Mailer>  MailerPtr;
    OneMult<Person,History>  HistoryPtr;

  public:
    Person()  {}
    ~Person()  {
      MailerPtr.unlink(this,0);
      HistoryPtr.unlink(this,0);
    }
    int checkIntegrity(){
      MailerPtr.checkIntegrity();
      HistoryPtr.checkIntegrity();
    }

    inline int link(Mailer *MailerObjPtr, int needBackTrace){
     return(MailerPtr.link(this,MailerObjPtr,needBackTrace));
     }
    inline int unlink(Mailer *MailerObjPtr,needBackTrace){
     return(MailerPtr.unlink(this,MailerObjPtr,needBackTrace));
     }
    inline int link(History *HistoryObjPtr, int needBackTrace){
     return(HistoryPtr.link(this,HistoryObjPtr,needBackTrace));
     }
    inline int unlink(History *HistoryObjPtr,needBackTrace){
     return(HistoryPtr.unlink(this,HistoryObjPtr,needBackTrace));
     }
};
```

```
class Dpgc:public Person{
    private:
      void Privateinterface() ;

    public:
      Dpgc()  {}

      ~Dpgc() {}

      int checkIntegrity(){
       Person::checkIntegrity();
      }

      void interface(){
        checkIntegrity();
        Privateinterface();
        }
};


class Doaa:public Person{
    private:
      void Privateinterface() ;
```

```
  public:
    Doaa()  {}

    ~Doaa()  {}

    int checkIntegrity(){
     Person::checkIntegrity();
    }

    void interface(){
      checkIntegrity();
      Privateinterface();
      }
};

class History{
    private:
      OneMult<History,Person>  PersonPtr;
      AgRangeMult<St_History,1>  St_HistoryPtr;
      AgRangeMult<Dept_History,1>  Dept_HistoryPtr;
      AgRangeMult<Faculty_Hist,1>  Faculty_HistPtr;
      AgRangeMult<Courses,1>  CoursesPtr;

    public:
      History()  {}

      ~History()  {
        PersonPtr.unlinkAll(this);
```

```
}

int checkIntegrity(){
  PersonPtr.checkIntegrity();
 St_HistoryPtr.checkIntegrity();
 Dept_HistoryPtr.checkIntegrity();
 Faculty_HistPtr.checkIntegrity();
 CoursesPtr.checkIntegrity();
}

inline int link(Person *PersonObjPtr, int needBackTrace){
 return(PersonPtr.link(this,PersonObjPtr,needBackTrace));
}
inline int unlink(Person *PersonObjPtr){
 return(PersonPtr.unlink(this,PersonObjPtr,needBackTrace));
}
inline int Ag_link(St_History *St_HistoryObjPtr){
 return(St_HistoryPtr.Ag_link(St_HistoryObjPtr));
 }
inline int Ag_unlink(St_History *St_HistoryObjPtr){
 return(St_HistoryPtr.Ag_unlink(St_HistoryObjPtr));
 }
inline int Ag_link(Dept_History *Dept_HistoryObjPtr){
 return(Dept_HistoryPtr.Ag_link(Dept_HistoryObjPtr));
 }
inline int Ag_unlink(Dept_History *Dept_HistoryObjPtr){
 return(Dept_HistoryPtr.Ag_unlink(Dept_HistoryObjPtr));
 }
```

```
      inline int Ag_link(Faculty_Hist *Faculty_HistObjPtr){
       return(Faculty_HistPtr.Ag_link(Faculty_HistObjPtr));
       }
      inline int Ag_unlink(Faculty_Hist *Faculty_HistObjPtr){
       return(Faculty_HistPtr.Ag_unlink(Faculty_HistObjPtr));
       }
      inline int Ag_link(Courses *CoursesObjPtr){
       return(CoursesPtr.Ag_link(CoursesObjPtr));
       }
      inline int Ag_unlink(Courses *CoursesObjPtr){
       return(CoursesPtr.Ag_unlink(CoursesObjPtr));
       }
};


class Rules{
    private:
     AgManyMult<Rule>  RulePtr;

    public:
      Rules()  {}

      ~Rules()  {
      }

      int checkIntegrity(){
       RulePtr.checkIntegrity();
      }
```

```
    inline int Ag_link(Rule *RuleObjPtr){
     return(RulePtr.Ag_link(RuleObjPtr));
     }
    inline int Ag_unlink(Rule *RuleObjPtr){
     return(RulePtr.Ag_unlink(RuleObjPtr));
     }
};
```

Appendix D

Template Library

```
#include <iostream.h>

//////////////////////////////////////////////////////////////////////
//
// class OneMult definition. This template maitains a pointer from
// left hand side class to right hand side class. This class will
// give the warning if any function tries to access the ``pointer"
// without initializing it i.e. it will preserve the multiplicty
// ``one". It has two functions called ``link" and ``unlink" to
// assign and delete the ``pointer" while maintaining ``referential
// integrity".
//
//////////////////////////////////////////////////////////////////////

template <class LhsClass,class RhsClass>
class OneMult {
 private:
```

```
 RhsClass   *Ptr;      //Actual pointer to the RHS class
 int present;          //if Ptr initialized present = 0 else =1

public:
 oneMult() {          //Constructor
     present = 0;     //Ptr not present
     }

 oneMult(RhsClass *rhs) {    //Constructor
     Ptr = rhs;       //Ptr is being assigned
     present = 1;     //Ptr present
     }

 ////////////////////////////////////////////////////////////
 // overload the operator "->" in such a way that it will
 // return Ptr object if present and it will give a WARNING
 // if the Ptr not present
 ////////////////////////////////////////////////////////////
 inline  RhsClass * operator->(){
       if(present == 1) return(Ptr);
       cout << " WARNING:Object not present";
       return(Ptr);
      }
 ////////////////////////////////////////////////////////////
 // The following method is a very useful one. One should call
 // this method before accessing any data_member of its class
 // if one of its data_members is oneMult type.
 ////////////////////////////////////////////////////////////
```

```
void checkIntegrity(){
   if(present == 0)
    cout << "\n WARNING:Object not present \n";
   return;
   }

/////////////////////////////////////////////////////////
// Following are the routines to link and unlink the other
// end participant. ''needBackTrace" variable tells whether
// it is neccessary to call link or unlink functions of the
// other end object. (i.e. update rhsPtr's link to this
// object.)
/////////////////////////////////////////////////////////


int link(LhsClass *This, RhsClass *rhsPtr,int needBackTrace){
    Ptr = rhsPtr;
    present = 1;
    if(needBackTrace == 1)
       rhsPtr->link(This,0);
    return(1);
    }

 int unlink(LhsClass *This,RhsClass *rhsPtr,int needBackTrace){
    if(present == 1 && rhsPtr == Ptr) {
                    if(needBackTrace == 1)
                       rhsPtr->unlink(This,0);
```

```
                  present = 0;
                  Ptr = NULL;
                  return(1);
                  }
    return(0);
    }

  void unlinkAll(LhsClass *This){
     present = 0;
     Ptr->unlink(This,0);
     }
};
```

```
//////////////////////////////////////////////////////////////////////
//
// class rangeMult definition. This template maitains a set of
// pointers from left hand side class to right hand side class.
// This class will give the warning if any function tries to access
// one of the pointers without initializing minimum number of pointers.
// i.e. it will  preserve the multiplicty "min+" eg:2+
//
//////////////////////////////////////////////////////////////////////

template <class LhsClass, class RhsClass, int minObjs>
class rangeMult {
 private:
  RhsClass *Ptr[100];//Actual array of pointers to the specified class.
                     // maximum of the range is 100;
  int noOfObjs;      // number of currently assigned Objects
  int minimum;

 public:
  rangeMult() {        //Constructor
      noOfObjs = 0;
      minimum = minObjs;
      }

   ////////////////////////////////////////////////////////////
   // overload the operator "[]" in such a way that it will
   // return Ptr object if minimum number of objects present.
   // and it will give a WARNING if not .
```

```
////////////////////////////////////////////////////////////
 inline RhsClass * operator[](int index) {
      if(noOfObjs >= minimum && index < noOfObjs)
                 return(Ptr[index]);
      cout << " WARNING:Object not present";
      return(Ptr[index]);
     }


////////////////////////////////////////////////////////////
// The following method is a very useful one. One should call
// this method before accessing any data_member of its class
// if one of its data_members is rangeMult type.
////////////////////////////////////////////////////////////
 void checkIntegrity(){
    if(noOfObjs < minimum)
     cout << " WARNING:Minimum number of Objects" <<
                        "(" << minimum << ") not present\n";

    return;
    }

 ////////////////////////////////////////////////////////////
 // Following are the routines to link and unlink the other
 // end participant. ''needBackTrace" variable acts as in the
 // OneMult case.
 ////////////////////////////////////////////////////////////
 int link(LhsClass *This, RhsClass *rhsPtr,int needBackTrace){
```

```
      Ptr[noOfObjs++] = rhsPtr;
      if(needBackTrace == 1)
        rhsPtr->link(This,0);
      return(1);
      }

  int unlink(LhsClass *This,RhsClass *rhsPtr,int needBackTrace){
     int i,j;
     for(i=0;i<noOfObjs;i++){
       if(rhsPtr == Ptr[i]) {
                       if(needBackTrace == 1)
                         rhsPtr->unlink(This,0);
                       for(j=i+1;j<noOfObjs;j++)
                        Ptr[j-1] = Ptr[j];  //Adjust Ptr array
                       noOfObjs -=1;
                       return(1);
                       }

        }
    return(0);
    }

  void unlinkAll(LhsClass *This){
     int i;
     for(i=0;i<noOfObjs;i++)
        Ptr[i]->unlink(This,0);
     noOfObjs = 0;
     }
};
```

```
//////////////////////////////////////////////////////////////////////
Like these two above template classes there are 4 more template
classes for association multiplicity implementation
   OptMult : to maintain ''Optional'' multiplicity
   ManyMult: to maintain ''Many'' multiplicity
   SetMult : to maintain ''Set'' multiplicity
   UniqMult: to maintain the qualifiers
//////////////////////////////////////////////////////////////////////
To maintain the multiplicity of the aggregation, there are
some templates which are same as association templates without
referential integrity
//////////////////////////////////////////////////////////////////////
These template class definitions are present in the file
  ''multiplicities.h''. This file is available with OOMTool.
```

Appendix E

Intermediate code format

This appendix gives the format of the intermediate code generated by OOMTool. Intermefiate code file contains sequence of records describing one of the OMT symbols in the diagram.
Each record starts with a number identifying the OMT symbol.
(e.g. 1: Association 2: Class 3: Generalization 4:Aggregation)

Fields in the Association record:

1. Type //always 1.
2. Name //Name of the association
3. class1 //first end class name of the association
4. Multy1 //first end multiplicity
5. role1 //first end role name
6. class2 //second end class name of the association
7. Multy2 //second end multiplicity
8. role2 //second end role name
9. link_attribute //link attribute class name

Fields in the Class record:

1. Type //always 2.
2. Name //Name of the Class
3. Attribute details

-number of types of attributes

-for each type

*type Name

*number of strings

*sequence of strings

4. Methods details(same as for attribute)

Fields in the Generalization record:

1. Type //always 3
2. Name //name of generalization
3. Generalized class name
4. number of subclasses
5. sequence of subclass names

Fields in the Aggregation record:

1. Type //always 4
2. Name //name of aggregation
3. Parent class name
4. parent multiplicity
5. number of subclasses
6. sequence of subclass names and multiplicities